

Building Resilient Cloud Over Unreliable Commodity Infrastructure

Piyus Kedia, Sorav Bansal
IIT Delhi

Deepak Deshpande, Sreekanth Iyer
IBM India Software Lab

Abstract

Cloud Computing has emerged as a successful computing paradigm for efficiently utilizing managed compute infrastructure such as high speed rack-mounted servers, connected with high speed networking, and reliable storage. Usually such infrastructure is dedicated, physically secured and has reliable power and networking infrastructure. However, much of our idle compute capacity is present in unmanaged infrastructure like idle desktops, lab machines, physically distant server machines, and laptops. We present a scheme to utilize this idle compute capacity on a best-effort basis and provide high availability even in face of failure of individual components or facilities.

We run virtual machines on the commodity infrastructure and present a cloud interface to our end users. The primary challenge is to maintain availability in the presence of node failures, network failures, and power failures. We run multiple copies of a Virtual Machine (VM) redundantly on geographically dispersed physical machines to achieve availability. If one of the running copies of a VM fails, we seamlessly switchover to another running copy. We use Virtual Machine Record/Replay capability to implement this redundancy and switchover. In current progress, we have implemented VM Record/Replay for uniprocessor machines over Linux/KVM and are currently working on VM Record/Replay on shared-memory multiprocessor machines. We report initial experimental results based on our implementation.

1. Introduction

Virtualization of server resources is the building block of cloud computing. Almost all cloud installations use dedicated high-performance server hardware resources with reliable and secure power, cooling and network infrastructure. Such installations are expensive, both at setup and operationally. On the other hand, much compute capacity in the form of idle desktops, lab machines, small server clusters, and other office computing equipment, remains underutilized. Under-utilization could be perennial (e.g., a machine that gets used mostly for lightweight browsing and/or work processing) or time-dependent (e.g., idle at night time or during summer months). Such scenarios are especially common in academic institutions (computing lab infrastructure, small high-performance compute clusters for individual research groups), software dev/test

environments (multiple desktops per user, or servers used to run builds and automated tests), and organizations with infrastructure across the world with different time zones. We attempt to utilize this unused capacity to support a cloud service, thus reducing wastage and lowering costs.

We run IaaS (infrastructure-as-a-service) cloud on these underutilized computers. The cloud virtual machines (VMs) are run on the underutilized computers in the background. We use hypervisor support on the host operating system to run these VMs without disrupting other ongoing activity. The cloud VM resource usage is maintained low enough to avoid any visible performance effects to the end-user of that computer. VMs are dynamically scheduled and migrated among the available computers appropriately, taking the underlying physical network topology into consideration. We call this model, *community cloud computing*, as it requires a community of users to allow usage of their underutilized resources. Because we use shared and relatively lower performance hardware to run the VMs, we do not expect to run performance-critical workloads on the community cloud. We expect the model to best suit long-running workloads which are less performance critical. e.g., compute-intensive scientific workloads or data-intensive analytic workloads. Such non-performance-critical workloads are often run on time-shared hardware in managed cloud environments to lower end-user costs; running them on community hardware (with less sharing) could provide similar or better performance at the same cost. Rough estimates of the underutilized compute capacity available in such unmanaged infrastructures indicate that this model, if successful, could result in significant cost savings.

As far as we can see, the primary challenge in achieving a successful implementation of a community cloud is maintaining reliability and availability, given that most of our physical infrastructure is unreliable. The user of a community cloud should expect availability similar to that on a regular cloud. In fact, migration of VMs between managed and unmanaged infrastructure should be opaque to the user — the user should simply expect the same level of performance and reliability irrespective of where the VM is running. Community-managed computers have unreliable power (no redundancy in power supplies, poor UPS support, users unplugging power cables in error, etc.), unreliable network connectivity (faulty network wires/NICs/hubs, users unplugging network cables, routers losing power supply, etc.), and unreliable storage (failing local disks, unreliable connections to NFS storage, etc.), among other things.

We propose the use of Virtual Machine Record/Replay to implement redundancy of live VM images (a similar approach is also used to implement high-availability services in commercial settings[7]). Each VM is simultaneously executed on n different physical machines, where n is the degree of redundancy. One of the VM replicas is designated the primary and executes on live network state, while the other $n - 1$ (secondary) replicas simply replay the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCEM '12 Oct 11-12, Bangalore, India.

Copyright © 2012 IEEE [to be supplied]...\$10.00

primary. On failure of the primary, one of the secondary replicas takes over. Because, the secondary replica was replaying the primary, its execution state is expected to be identical (or very close) to that of the primary at failure time. The number of secondary replicas can be always maintained at $n - 1$ (barring the short intervals of time after failure of either primary or secondary replicas, when a new secondary needs to be spawned). The choice of the different physical machines which execute the replicas of one VM can be made taking the network and power supply topology into consideration (e.g., failure of one network router should not cause all replicas to become disconnected). We expect a redundancy factor of 3-5 ($n=3-5$) to provide acceptable reliability for our cloud service.

There are two primary challenges in realizing a practical implementation of our model:

1. *Failover based on record/replay*: We need an efficient record/replay and failover mechanism. We implement uniprocessor record/replay on Linux/KVM and discuss its performance in detail. The performance of our implementation is comparable to that of VMware's commercial closed-source workstation product[8]. For seamless network operations on failover, we require that all VM replicas belong to the same VLAN. On primary failure, a secondary replica becomes the primary and the network routing is automatically adjusted using ARP network protocol. At failover time, the secondary could be slightly behind the primary (e.g., due to network latency of streaming the record log), and this could result in application-level inconsistencies over the network. Fortunately, we find that this happens rarely in practice. We discuss our failover experiments and findings in detail.
2. *Scheduling*: The replicas of multiple VMs need to be scheduled (placed) across multiple physical machines with the following constraints:
 - VMs should only be run on under-utilized hosts. If a host shows high utilization, the VM should be migrated to another host.
 - Different replicas should be placed across different physical machines and preferably on different physical network branches (to guard against router failures)
 - The VM image store should always be available to all VM replicas. This could be either done using a reliable NAS server or using a distributed (and replicated) filesystem. With a NAS server, VM placement is further constrained to be on hosts close to the storage, to minimize network traffic. For a distributed filesystem using commodity disks on the community computers, efficient algorithms need to be devised to place the storage redundantly such that it guards against failure and still remains accessible to the computing hosts without too much network traffic.
 - Finally, the replica placement must remain sensitive to the network traffic caused due to the streaming of record logs between primary and secondaries.

These scheduling constraints are difficult, and require significant experimentation. We have not yet fully explored the solution space to this problem in this paper. We provide an initial assessment of the tradeoffs involved, and intend to study them experimentally in detail in future.

There are other alternatives to VM record/replay for implementing redundancy, such as periodic snapshotting of VM state or file system record/replay. The choice of approach depends on the trade-off between the performance overhead during normal operation, and the nature of state loss on failure. VM Record/Replay oper-

ates on 5-10% runtime overhead while maintaining a loss of typically a few 100 milliseconds of execution state. On the other hand, file system record/replay techniques have lower runtime overhead but require a full VM restart/reboot on failure (resulting in an execution state loss of 10s of seconds). Periodic snapshotting of VM state also suffers from the drawback of loss of large execution state on failure. We believe that record/replay has the best performance and reliability tradeoffs among the available alternatives.

The paper is organized as follows. Section 2 discusses our implementation of VM record/replay and failover on Linux/KVM. Section 3 discusses our experiments and results, Section 4 discusses related work, and Section 5 concludes.

2. Virtual Machine Record/Replay

ReTrace[9] demonstrated the capability to record/replay (R/R) an execution in VMware Workstation and reported as low as 5% runtime overhead, and 0.5 byte per thousand instructions log growth rate. VM R/R works by recording all external input to virtualized devices and the timing of interrupts as they are delivered to the guest. Time is counted by counting the number of instructions (or branches) executed by the guest. On x86 platforms, the three tuple (`nbranches`, `rip`, `rcx`) uniquely identifies a logical guest execution epoch, where `nbranches` is the number of branches executed by the guest, `rip` is the guest's current instruction pointer, and `rcx` is the current value of count register (needed for instructions with `rep` prefix). The `nbranches` counter is maintained using hardware performance counters. All deterministic instructions (i.e., instructions producing identical output on same input irrespective of time of execution) can be executed unmodified directly on hardware. All non-deterministic instructions must be made to trap to the Virtual Machine Monitor (VMM) and their non-deterministic result recorded (e.g., `rdtsc` on x86). Because a huge fraction of executed instructions in common workloads are deterministic (e.g., >99% for most compute-intensive workloads), the overhead of VM R/R is minimal. All interrupts delivered to the guest are recorded along with their epoch (`nbranches`, `rip`, `rcx`). Emulated devices are recorded by recording all the non-determinism in the device emulation code. For example, if the device uses an external input (e.g., network packets), that input is logged.

During replay, the results of non-deterministic instructions and non-deterministic device inputs are obtained from the log. Replying interrupts requires special care. We require the guest to yield control to the VMM at the interrupt epoch (`nbranches`, `rip`, `rcx`) for accurate delivery of the replayed interrupt. On x86 architectures, this can be achieved by configuring the performance counters to overflow at the desired branch count `nbranches`, which generates an interrupt causing the guest to yield control to the VMM. We then single-step the guest till we reach the desired (`rip`, `rcx`) before injecting the replayed interrupt to the guest. On current x86 architectures, this interrupt-on-overflow mechanism for performance counters is imprecise. It is possible for the branch count to overshoot the desired value by up to 128 before an interrupt is generated. The solution to this problem (as also noted in [3]) is to configure the performance counters to generate an interrupt at (`nbranches - 128`) and then single-step the guest till the interrupt injection epoch (`nbranches`, `rip`, `rcx`). This careful singlestepping near interrupt injections causes extra runtime overhead during replay (compared to record).

There are two alternatives while replaying the virtual disk device, namely *full-replayed disk* or *output-replayed disk*. A full-replayed disk is snapshotted at the start of the recording session so that its state remains reproducible at any execution epoch. The reads/writes to the disk do not need to be recorded, as they will be identical during record and replay, provided all other external and timing related non-deterministic inputs remain identical. For

an output-replayed disk, the disk device is not snapshotted and only the values returned by the disk on every disk read are recorded. Here, the disk state cannot be reconstructed but the CPU/memory state can still be reconstructed at any execution epoch. We use full-replayed disks for our experiments.

We implement redundancy using one primary and multiple secondary VM replicas, all running on different physical hosts simultaneously. The primary records and all secondaries replay. The record log (also called execution trace) is streamed from the primary to the secondary using TCP/IP. On failure, we switch primary to one of the secondaries.

We discuss the performance of our VM R/R implementation and the failover mechanism in the next section.

3. Experimental Results

We implemented KVM R/R inside Linux kernel 2.6.36.4. Our implementation uses x86 hardware performance counters to count branches, and the monitor trap flag (MTF) for single-stepping. We implemented record/replay functionality for emulated devices inside Qemu. Our implementation is complete enough to run a full Linux guest. We tested our implementation with all types of applications including graphical and networked applications. For example, we could successfully view a Youtube video while running record/replay underneath. We tested the stability of our implementation by running it continuously for over 24 hours with an active guest.

We used Qemu 0.13.0 with default configuration for device emulation record/replay. Our 32-bit Gentoo Linux guest was configured with fully-emulated e1000 network and IDE disk devices. The guest ran with 128MB physical memory, 512MB swap space, 8GB disk space and other devices emulated by Qemu by default. We ran our experiments on a machine with 12 Intel Xeon X5650 2.67 GHz processor cores, 24GB memory, and 300GB disk.

We evaluate our implementation using the benchmarks listed in Table 1. The benchmarks have been chosen to stress different components of the virtual machine monitor. Some of the benchmarks have been inspired by a previous VMM performance study[5]. The table also presents the typical logsize growth rate of each benchmark. The two logsize growth columns are for different ways of recording the disk device, namely output replay (records and replays the output of the disk device) and full replay (emulates the disk device fully). Figure 1 shows the performance characteristics of our VM R/R implementation on KVM. We show results compared to KVM as baseline. We deliberately don't show results compared to native execution, because performance difference between KVM and native execution is either negligible (for compute-intensive workloads) or is heavily dependent on the chosen virtual devices (for I/O-intensive workloads). We simply use the default KVM/Qemu virtual devices.

The `emptyloop` benchmark is compute-intensive and represents all compute-intensive workloads executing at user privilege level; `gpid` exercises the system call handling mechanism in Linux; `forkbomb` exercises the process creation and destruction methods (including page table manipulations); `cp` exercises the disk; `inet` and `onet` stress the network; and `sleep` emulates an idle system. `iscp` exercises both network and disk, while `lincompile` is a macrobenchmark that combines CPU, memory, and disk usage. We measure performance using the host's wall clock time. Figure 1 shows the performance of VM Record (`kvm-record`) and VM Replay (`kvm-replay`).

We use full-replayed disks for our R/R experiments. Each runtime is divided into the percentage of time spent in the guest, in the host kernel, and other activities including I/O and idle time. The overhead of recording is within 20% for all our benchmarks. The performance of VM Replay can be worse due to single-

stepping. The overhead of VM R/R on compute-intensive applications (`emptyloop` and `gpid`) is almost zero. The overhead is more for I/O intensive applications such as `cp` and `iscp`. `forkbomb` has higher R/R overhead than `emptyloop` due to paging activity and swap disk usage. The performance of `sleep` on `kvm-replay` is faster than on `kvm` because executions of the x86 `halt` instruction by the guest's idle thread finish instantaneously on `kvm-replay`, while they wait for an external interrupt on `kvm` and other variants. Compute-intensive applications spend almost all their time in the guest, while I/O intensive applications spend a significant fraction of time in the host user-level code for device emulation.

Our performance results indicate that recording and replaying incurs only small overheads for uniprocessor VMs and it is thus feasible to enable recording on user-facing cloud VMs. Our results also indicate that replaying could sometimes be slower than recording. The difference in performance is due to the hardware support for branch counters and has implications while running VMs redundantly in primary/secondary mode. If replay is slower, the replaying process could slowly *drift* from the recording process causing the replaying process to be much behind after a long time. This drift can be upper-bounded by forcing the record process to wait for the replay process after the drift exceeds a limit. This can result in further undesirable loss of performance (in our experiments, we observe 10-20% performance loss on compute-intensive workloads). The performance loss is only seen on compute-intensive workloads. The drift value also indicates the magnitude of the "loss of execution state" in the event of a primary failure, because the logical (user-visible) state will appear to have rolled back by the drift value. Figure 2 plots the drift over time for different workloads. Only `emptyloop` results in high drift values over time.

Failure of the primary replica could potentially result in network-level inconsistencies due to the rollback in the logical state of the VM. Because almost all network protocols are resistant to host failures, such inconsistencies are usually recoverable. We performed experiments to measure this effect. We ran connection-less `ping` and `http` servers on our VM and accessed them through a remote client. We then repeatedly failed the primary replica and observed that both `ping` and `http` requests continued after small intervals of unresponsiveness. We next ran a long connection-oriented `scp` session (`iscp`) on our VM and caused the primary replica to fail while the session was active. On failure, the `scp` session stalled. Analysis of network traffic trace revealed that a rollback caused the TCP sequence numbers to be rolled back which confused the remote client which had previously observed packets with higher sequence numbers.

We propose the use of *delayed network sends* to deal with this problem. Here, network packets originating from the primary replica are buffered (and not *released* to the external world) till a secondary replica has not replayed the send of the network packet. After the secondary replica has reached the execution epoch of the packet send, the packet is actually released into the external network. If a failure occurs, the network buffer is discarded. This ensures that no external inconsistencies occur on primary failure. We ran `iscp` under this delayed network send configuration and confirmed that the session continued uninterrupted (although with temporary performance effects) even on failure.

Delayed network sends have effects on the network latency and throughput, however. To measure this, we configured a primary and a secondary with delayed network sends and measured latency and bandwidth with increasing drift between the primary and secondary. We measure drift using the number of branches executed by the guest, which is an indirect but more deterministic measure of time. Figures 3 and 4 plot our results. The ping latency (Figure 3) increases by around 12ms for every increase of 1000 branches in

Benchmark	Description	Log Growth (full) KB/sec	Log Growth (output) KB/sec
emptyloop	A process running a compute-intensive loop	32.3	32.5
gpid	A process repeatedly calling the <code>getpid</code> system call on Linux	33.0	33.1
forkbomb	Forks 1 million processes, each of which exits gracefully	65.8	89.2
cp	Copies a 100MB file within the same directory on Linux ext3	10.6	5080
inet	Receives data over TCP socket in 4-byte chunks	793	791
onet	Sends data over TCP socket in 4-byte chunks	714	732
sleep	Calls <code>sleep(10)</code> (idles for 10 seconds)	30.2	32.1
iscp	Copies a 100MB file from the host machine to the guest VM using <code>scp</code> over the emulated network card at maximum rate	755	732
lincompile	Builds the Linux kernel from source	42.7	195

Table 1. Benchmark description and logsize growth characteristics for full-replayed and output-replayed disks.

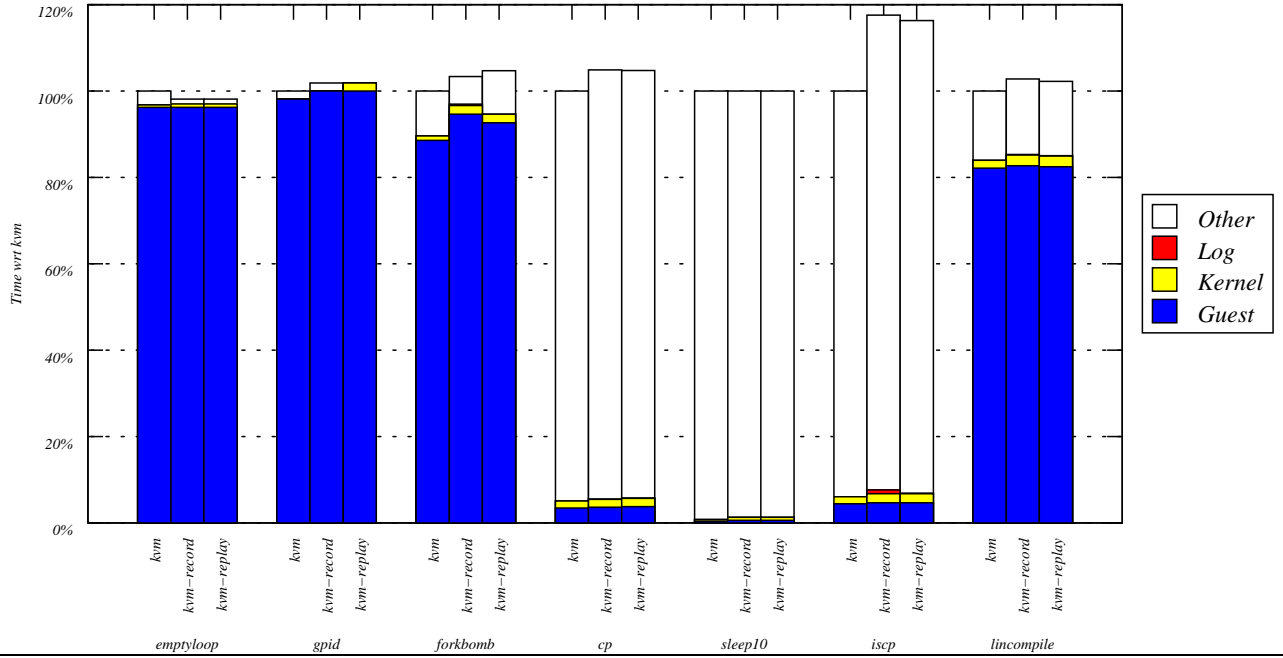


Figure 1. Performance of KVM Record/Replay.

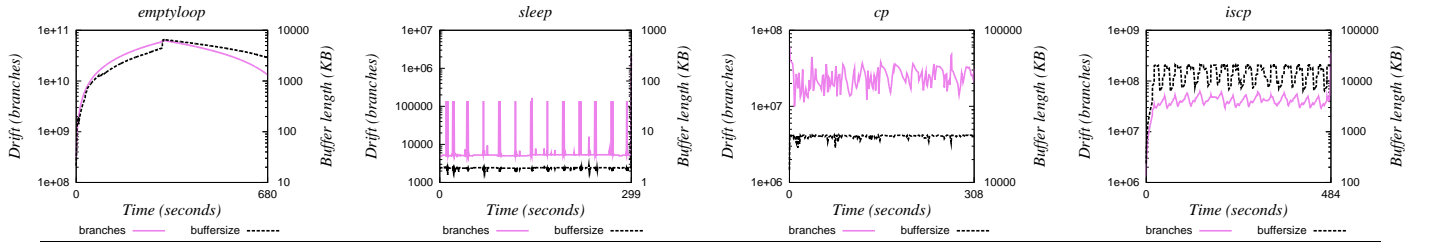


Figure 2. Distribution of drift across time for `emptyloop`, `sleep`, `cp`, and `iscp`.

the rollback window. The effect of drift on the completion time of an `iscp` session is less pronounced (Figure 4).

We also measured failover latencies. In our initial experiments, a secondary VM takes over as primary within a few seconds of the primary failure. We intend to perform more detailed experiments measuring failover latencies in future.

4. Related Work

Previous efforts on utilizing idle compute capacity include SETI@Home[2], Folding@Home[4], BOINC project[1], etc. Our work has a similar philosophy. The difference is in the level of abstraction. These previous efforts require that the programs be written to a specific programming model, and then provide a middleware which needs to be installed in all participating host machines. The middleware then coordinates and schedules the client programs. In contrast,

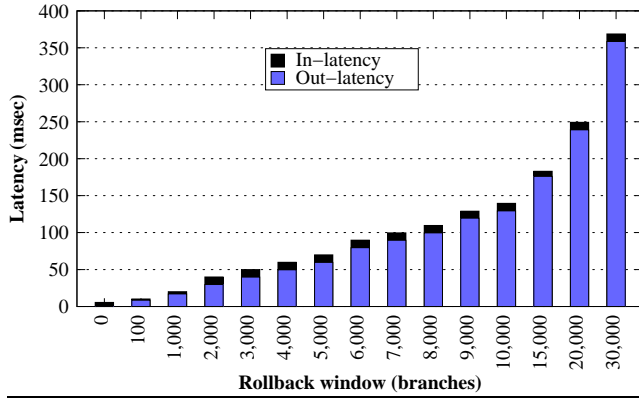


Figure 3. Ping latency with increasing rollback window size on *kvm-mrmd*

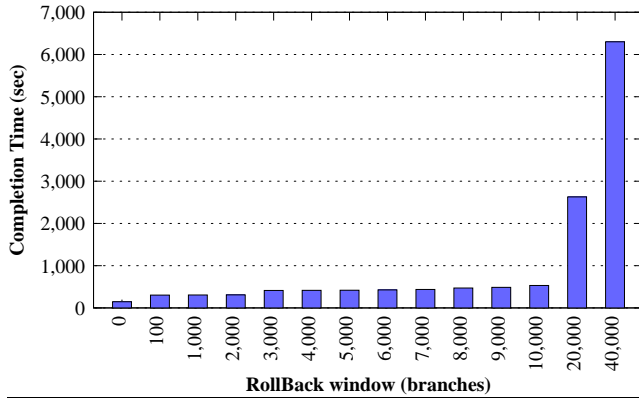


Figure 4. Completion time for *iscp* with increasing rollback window size on *kvm-mrmd*

our abstraction is more general (and often more powerful) than the middleware approach. Our computation units are VMs, allowing a client full freedom to run her favourite OS and applications on the participating hosts, without compromising security and reliability. To provide reliability, the middleware-based approaches usually constrain the programming model for easy restartability. In contrast, we allow a completely flexible programming model and provide reliability through efficient recording and replaying.

Our current prototype can efficiently record and replay a uniprocessor VM. We have also implemented record/replay for multiprocessor VMs. Multiprocessor record/replay is significantly harder due to the presence of race conditions on shared memory by multiple processors. We have implemented a page-ownership scheme based on CREW (concurrent read exclusive write) protocol[3] to record and replay a guest OS. We can successfully replay an unmodified guest, albeit at high overheads. The overheads depend on the workload and could be as high as 2-3x slowdowns for 2-processor VMs. Another approach, DoublePlay[6], has been proposed to make multiprocessor record/replay more performant. DoublePlay works by recording the order of all synchronization operations in the program being recorded. Because a guest OS could have arbitrary synchronization primitives, it is hard to directly use DoublePlay's ideas for VM Record/Replay. We are currently working on approaches to make multiprocessor VM record/replay faster.

5. Conclusion and Future Work

We present our initial results on our efforts to develop a community cloud. We have implemented record/replay inside Linux/KVM and measure its performance on various workloads. We also analyze concurrent executions of the record and replay processes to understand their interplay. In future, we plan to design and test scheduling algorithms for the primary and secondary replicas of the cloud VMs. We are also working on realizing efficient multiprocessor VM record/replay.

6. Acknowledgment

The authors would like to thank IBM for supporting this work in part by grant under the CAS Program.

References

- [1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04.
- [2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11), Nov. 2002.
- [3] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2008.
- [4] S. M. Larson, C. D. Snow, M. Shirts, V. S. P. and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology.
- [5] Performance Aspects of x86 Virtualization by Ole Agesen. Performance aspects of x86 virtualization by Ole Agesen. <http://www.vmworld.com/docs/D0C-2476>.
- [6] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS'11*, pages 15–26, 2011.
- [7] VMware High Availability. VMware High Availability. <http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf>.
- [8] VMware Workstation. VMware Workstation. <http://www.vmware.com/products/workstation>.
- [9] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, volume 3, 2007.